COMP61232 Matrix Multiplication- Neon

Kunjian Song

1 Task 1

1.1 Code Study and Cycle Estimation – Matrix Multiplication Basic Function

Most of the modern processors are multi-issue processors, capable of issuing multiple instructions with microarchitecture designs that deals with both data and control dependencies. Hence, estimating the total cycles used to complete a task can be challenging without studying the details of microarchitecture of the target chip, and varying depending on the assumptions made.

Since Cortex-A53 is Load-Store architecture. The main strategy to estimate the cycles is sum up cycles used in the following two area:

- Data loading
- Data computation

Therefore, the cycle estimation in this section was purely derived based on the Fact and Assumption tables of the microarchitecture of Coretex-A53 being used in Raspberry Pi 3 BCM2837 processor, rather than following the conventional way using the formula "Cycle Count = cycle/iteration * iterations". Any changes in the assumption could lead to a potentially noticeable difference in the final estimation result.

The known facts of Cortex-A53 that can influence the cycle estimation are listed in the Fact table below.

Facts of Cortex-A53	Impact on Matrix Multiplication Computation
RISC Load-Store architecture.	No direct memory address mode can be used in the instruction. The functional units take their operands from registers, not from memory. If the requested data is not in the cache, memory reference is expensive.
In-Order Execution	It's the in-order core in ARM big.Little architecture. If an instruction is waiting for dependencies in a pipeline, it stalls the pipeline.
32 bit (as used for our experiment on Raspberry Pi 3)	Each integer is 4 bytes.
64 Byte cache line	One cache line can hold 16 integers. As for each row in a 1024x1024 matrix, it needs 64 cache lines to hold all 1024 integers in a row. Each matrix would need 65536 cache lines to hold all data.
L1d cache is 16KB	L1d cache can hold a maximum of 250 cache lines, approximately 4 rows of the experiment matrix.
L2 cache is 512KB	L2 cache can hold a maximum of 8000 cache line.

Table 1: Facts of Cortex-A53 and its impact on matrix multiplication

From Table 1, the matrix multiplication basic function will stress the memory system a lot as each matrix will need 65536 cache lines. Even L2 in this case cannot hold a single full matrix, and there are three of them – matrixA, matrix and matrix C.

Since the multi-dimensional array in C is row-major. The memory layout of a matrix is like:

A[0][0]
A[0][1]
…
A[0][1023]
A[1][0]
A[1][1]
…
A[1][1023]
A[2][0]
…
A[1023][1022]
A[1023][1023]

Figure 1: Memory layout of a matrix programmed in C

The data in each cache line is ordered and contiguous as in per row. This will give a huge bottleneck when computing matrix multiplication in which the data in second matrix is accessed per row at a time, i.e. from a different cache line because each cache line in Cortex-A53 can only hold 16 integers, far less than the 1024 integers per row. Hence the underlined part in the code snippet below is the bottleneck of the basic matrix multiplication function.

```
void matrix_multiply_basic()
{
    int i,j,k;
    for (i = 0 ; i < N ; i+=1)
        for(j = 0 ; j < N ; j +=1)
            for(k = 0 ; k < N ; k +=1)
                matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
}
```

Figure 2: Code snippet of the basic matrix multiplication function. Underlined part is the bottleneck.

The underlined part “matrixB[k][j]” could cause lots of cache misses when k increases every time in the most inner loop. Each matrix requires 65536 cache lines to hold all the data. Hence “matrixB[k][j]” could stress the cache line replacement policy and eviction mechanism in both L1d and L2. This would have a huge impact on the Cycle Per Instruction (CPI) value if the core wastes cycles waiting for the data to be loaded.

The assumptions of the microarchitecture are listed in the Assumption Table (Table 2) for cycle estimation.

Assumptions	Impact on Total Cycle Estimation
Single issue, 1 ALU pipeline	To reduce complexity for estimation. If multi-issue and multiple ALU pipelines are take into account, the estimation would be more complex if considering the number of read/write ports of the register files, arbitration of request of in cache, resource dependencies. It's not accurate but good for estimation.
Write-back cache	No cycles wasted to maintain the cleanness of data.
8 stage pipeline and CPI = 5 cycles	Assume CPI value is 5. (Ideally it should be 1 if no stall happens)
ONE thread execution finished on ONE core	There will be no multi-core execution. No effort or cycles for core switching.
Read only on A's and B's cache line	Read is easy to handle.
Read with write intention for C's cache line	No effort or cycles for cache coherency.
Low branch misprediction rate of instruction fetch unit	The cycles wasted to flush the pipeline in this case can be ignored.
L1's victim cache is stored in L2, and L2 cache line eviction is smart enough to keep some matrixB's cache line for future usage.	When accessing matrix[k][j] as k is increasing, the core does not need to access memory everytime. More importantly no cycles wasted for searching the hit way (associativity) in L2. Otherwise it would be too complicated to do the cycle estimation.
Access L1 coses 3 cycles, L2 costs 30 cycles. Access main memory costs 500 cycles.	The actual cycles could be more.
No data prefetching	Otherwise it would be too complicated to estimate if taking this into account. The actual execution could have taken some advantage of this, which means the estimation might more than needed.

Table 2: microarchitectural assumption for cycle estimation

The matrixB's cache lines held in L1d, the quicker the program can run. Hence, assuming that optimized usage of L1d usage to overcome the bottleneck of accessing matrixB are listed as below:

- Assume the compiler and instruction issue can optimise the usage of L1d
- It knows B's cache line has low temporal locality. But A and C has high spacial locality.
- When a B's cache line is evicted from L1, L2 knows its temporal locality.
- B's cache line in L1d is only read once, because the next k iteration will need the data from another row which is (1024 * 4) bytes further away in the memory, i.e. another cache line.
- A's and C's cache line in L1d are read repeatedly, i.e. high spacial locality. All A's and C's cache line are read from memory, and their evictions from L1d are not held in L2.
- The key is B's cache line here. That's the bottleneck.
- L2 is all used to hold B's CL evicted from L1 – most optimised based on the teomporal locality.
- L1d detailed usage:
    - Only 192 slots in L1d are assigned to the test program – Matrix Multiplication
    - 64x A's CL (a row) + 64x C's CL (a row) + 64x B's CL (a column)
- L2 usage: L2 knows to keep B's cache line evicted from L1.

The cycle estimation is done through the following steps, group by two parts – Data Computation and Loading.

Cycles used for Data Computation:

- Computation: target rows are in L1d, each element in C requires 4 cycles for computation, hence 2x ALU, 1024*4 / 2 = 2048 cycles

Cycles used for Data Loading:

- Load data to L1d:
  - Load data B:
    - 65536 cache lines for B:
      - First time access (from memory): 65536 * 500 = 32768000 cycles
      - L2 can hold maximum of 8000 cache lines, hence need ~8 times to hold all 65536 cache lines.
  - Load data A and C:
    - First time access each cache line (from memory): 655636 * 500 * 2 = 65536000 cycles
- Load data from L1d:
  - Load integers from A/B/C:
    - 1024 * 3 * 2 = 6144 cycles
- In total (sum of underlined numbers above):
  - 2048 +32768000 + 65536000 +6144 = 98312192 cycles

1.2 Compile and Run – Matrix Multiplication Basic Function

The results of matrix multiplication basic function are shown in Table 3.

O-level Optimization	Perf_Counter	T
0	250275769905	208.806454
1	204048086042	170.547883
2	201503191173	168.807444
3	201459459227	168.768972

Table 3: Execution time and used instruction counter.

Table 3 shows that O3 optimization level really reduces the execution time and the cycles used to finish the task.

1.3 Comparison of O0 and O3 Disassembly - Matrix Multiplication Basic Function

This section compares the disassembly codes of matrix_multiply_basic function between O0 and O3. The code snippets are shown in Figure 3 and Figure 4.

```
000108a0 <matrix_multiply_basic>:
   108a0:       e52db004        push    {fp}            ; (str fp, [sp, #-4]!
   108a4:       e28db000        add     fp, sp, #0
   108a8:       e24dd014        sub     sp, sp, #20
   108ac:       e3a03000        mov     r3, #0
   108b0:       e50b3008        str     r3, [fp, #-8]
   108b4:       ea00002e        b       10974 <matrix_multiply_basic+0xd4>
   108b8:       e3a03000        mov     r3, #0
   108bc:       e50b300c        str     r3, [fp, #-12]
   108c0:       ea000025        b       1095c <matrix_multiply_basic+0xbc>
   108c4:       e3a03000        mov     r3, #0
   108c8:       e50b3010        str     r3, [fp, #-16]
   108cc:       ea00001c        b       10944 <matrix_multiply_basic+0xa4>
   108d0:       e59f10b8        ldr     r1, [pc, #184]  ; 10990 <matrix_multi
   108d4:       e51b3008        ldr     r3, [fp, #-8]
   108d8:       e1a02503        lsl     r2, r3, #10
   108dc:       e51b300c        ldr     r3, [fp, #-12]
   108e0:       e0823003        add     r3, r2, r3
   108e4:       e7912103        ldr     r2, [r1, r3, lsl #2]
   108e8:       e59f00a4        ldr     r0, [pc, #164]  ; 10994 <matrix_multi
   108ec:       e51b3008        ldr     r3, [fp, #-8]
   108f0:       e1a01503        lsl     r1, r3, #10
   108f4:       e51b3010        ldr     r3, [fp, #-16]
   108f8:       e0813003        add     r3, r1, r3
   108fc:       e7903103        ldr     r3, [r0, r3, lsl #2]
   10900:       e59fc090        ldr     ip, [pc, #144]  ; 10998 <matrix_multi
   10904:       e51b1010        ldr     r1, [fp, #-16]
   10908:       e1a00501        lsl     r0, r1, #10
   1090c:       e51b100c        ldr     r1, [fp, #-12]
   10910:       e0801001        add     r1, r0, r1
   10914:       e79c1101        ldr     r1, [ip, r1, lsl #2]
   10918:       e0030391        mul     r3, r1, r3
   1091c:       e0822003        add     r2, r2, r3
   10920:       e59f0068        ldr     r0, [pc, #104]  ; 10990 <matrix_multi
   10924:       e51b3008        ldr     r3, [fp, #-8]
   10928:       e1a01503        lsl     r1, r3, #10
   1092c:       e51b300c        ldr     r3, [fp, #-12]
   10930:       e0813003        add     r3, r1, r3
   10934:       e7802103        str     r2, [r0, r3, lsl #2]
   10938:       e51b3010        ldr     r3, [fp, #-16]
   1093c:       e2833001        add     r3, r3, #1
   10940:       e50b3010        str     r3, [fp, #-16]
   10944:       e51b3010        ldr     r3, [fp, #-16]
   10948:       e3530b01        cmp     r3, #1024       ; 0x400
   1094c:       baffffdf        blt     108d0 <matrix_multiply_basic+0x30>
   10950:       e51b300c        ldr     r3, [fp, #-12]
   10954:       e2833001        add     r3, r3, #1
   10958:       e50b300c        str     r3, [fp, #-12]
   1095c:       e51b300c        ldr     r3, [fp, #-12]
   10960:       e3530b01        cmp     r3, #1024       ; 0x400
   10964:       baffffd6        blt     108c4 <matrix_multiply_basic+0x24>
   10968:       e51b3008        ldr     r3, [fp, #-8]
   1096c:       e2833001        add     r3, r3, #1
   10970:       e50b3008        str     r3, [fp, #-8]
   10974:       e51b3008        ldr     r3, [fp, #-8]
   10978:       e3530b01        cmp     r3, #1024       ; 0x400
   1097c:       baffffcd        blt     108b8 <matrix_multiply_basic+0x18>
   10980:       e1a00000        nop                     ; (mov r0, r0)
   10984:       e28bd000        add     sp, fp, #0
   10988:       e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
   1098c:       e12fff1e        bx      lr
   10990:       00821044        .word   0x00821044
   10994:       00021044        .word   0x00021044
   10998:       00421044        .word   0x00421044
```

Figure 3: Disassembly code snippet generated by O0

```
00010878 <matrix_multiply_basic>:
   10878:       e92d41f0        push    {r4, r5, r6, r7, r8, lr}
   1087c:       e59f6058        ldr     r6, [pc, #88]   ; 108dc <matrix_mul
   10880:       e59fe058        ldr     lr, [pc, #88]   ; 108e0 <matrix_mul
   10884:       e2868501        add     r8, r6, #4194304        ; 0x400000
   10888:       e59f5054        ldr     r5, [pc, #84]   ; 108e4 <matrix_mul
   1088c:       e2464a01        sub     r4, r6, #4096   ; 0x1000
   10890:       e24e7a01        sub     r7, lr, #4096   ; 0x1000
   10894:       e5941004        ldr     r1, [r4, #4]
   10898:       e1a03007        mov     r3, r7
   1089c:       e1a02005        mov     r2, r5
   108a0:       e5b30004        ldr     r0, [r3, #4]!
   108a4:       e592c000        ldr     ip, [r2]
   108a8:       e153000e        cmp     r3, lr
   108ac:       e2822a01        add     r2, r2, #4096   ; 0x1000
   108b0:       e021109c        mla     r1, ip, r0, r1
   108b4:       1afffff9        bne     108a0 <matrix_multiply_basic+0x28>
   108b8:       e5a41004        str     r1, [r4, #4]!
   108bc:       e1540006        cmp     r4, r6
   108c0:       e2855004        add     r5, r5, #4
   108c4:       1afffff2        bne     10894 <matrix_multiply_basic+0x1c>
   108c8:       e2846a01        add     r6, r4, #4096   ; 0x1000
   108cc:       e1560008        cmp     r6, r8
   108d0:       e283ea01        add     lr, r3, #4096   ; 0x1000
   108d4:       1affffeb        bne     10888 <matrix_multiply_basic+0x10>
   108d8:       e8bd81f0        pop     {r4, r5, r6, r7, r8, pc}
   108dc:       00022040        .word   0x00022040
   108e0:       00422040        .word   0x00422040
   108e4:       00821044        .word   0x00821044
```

Figure 4: Dsiassembly code snippet generated by O3

The difference between O3 and O0 are summarized below:

- Overall more instructions are generated by O0. O3 can generate more concise block.
- For O0, compiler only utilized r0- r3 and do the calculation piece by piece, while compiler O3 uses more registers, {r1 – r8}.
- For O0, it's more like a direct "translation" of the code, using multiple conditional branch instruction (marked in green in Figure 3), while compiler O3 uses less branching which could have potentially reduce the branch misprediction and eases the instruction prefetching.
- For O0, it has lots of data dependencies (Read-After-Write), as marked in red in Figure 3, while O3 removes these dependencies by using more registers.
- For O0, it uses lots of LDR instruction (as marked in blue in Figure 3). This could stress the cache memory syste,/
- For O3, it uses more advanced instruction, MLA – multiply-accumulate.

To sum up, O0 in general just did a plain loop implementation, lots of load operations, conditional branching – stresses both memory system and instruction fetching. O3 in general uses less load operations, less branching, more advanced instruction (MLA) – less stress on memory system and less branching. O3 is not just a "straightforward" translation of the original loop. Overall, compiler did a good job in optimizing.

1.4 Inline Assembly Implementation - Matrix Multiplication Basic Function

The inline assembly implementation is shown in Figure 5. It's shorter than the O0 disassembly. It utilized {r0-12, r14} registers, more than O0 and O3. The I, J and K loops are implemented in line 73, 78 and 86. The main idea here is to locate the target A's row (line 82) and target B's column (line 84) and do the calculations.

```
    asm volatile(
        "push {r0-r12, r14}"            "\n\t"  //s
                                      // initialise b
        "mov r3, %[matrixA_addr]"       "\n\t" // m
        "mov r4, %[matrixB_addr]"       "\n\t" // m
        "mov r5, %[matrixC_addr]"       "\n\t" // m
        "mov r12, #4096"                "\n\t" // f
        "mov r14, #4"                   "\n\t" // f

        "mov r0, #0"                    "\n\t" // i
    "I_LOOP:"                           "\n\t"
        "cmp r0, #1024"                 "\n\t" // H
        "beq END"                       "\n\t" //
                                               //
        "mov r1, #0"                    "\n\t" // j
    "J_LOOP:"                           "\n\t"
        "cmp r1, #1024"                 "\n\t" // H
        "beq NEXT_I_ITERATION"          "\n\t" //
                                               //
        "mla r8, r12, r0, r3"           "\n\t" // g
        "mov r2, #0"                    "\n\t" // k
        "mla r10, r14, r1, r4"          "\n\t" // g
        "mov r9, #0"                    "\n\t" // i
    "K_LOOP:"                           "\n\t"
        "ldr r6, [r8]"                  "\n\t" // l
        "ldr r7, [r10]"                 "\n\t" // l
        "mla r9, r6, r7, r9"            "\n\t" // r
                                      // move A and B
        "add r8, r8, r14"               "\n\t" // n
        "add r10, r10, r12"             "\n\t" // n
                                      // k + 1
        "add r2, r2, #1"                "\n\t" // k
        "cmp r2, #1024"                 "\n\t" // H
        "bne K_LOOP"                    "\n\t" //
        "add r1, r1, #1"                "\n\t" //
        "str r9, [r5], #4"              "\n\t" //
        "b J_LOOP"                      "\n\t"

    "NEXT_I_ITERATION:"                 "\n\t"
        "add r0, r0, #1"                "\n\t" // i
        "b I_LOOP"                      "\n\t" // g
    "END:"                              "\n\t"
        "pop {r0-r12, r14}"             "\n\t" //re
```

Figure 5: Inline assembly implementation of basic matrix multiplication

1.5 Inline Assembly Performance

The execution results are shown below:

Perf_Counter	T
203273387152	170.284824

Table 4: execution results of inline assembly

Note that there was no optimization level experiments for inline assembly, as compiler just takes what's in that assembly and place it in the object file. The result in this section will be plotted in one figure in section 2.5.

2 Task 2 – NEON

2.1 Cycle Estimation Using NEON

NEON has 16 quadword (32 byte) registers {q10 – q15}, each of them mapping to two doubleword registers. To estimate the cycles, the first step is to calculate how workload can be executed in parallel. To fully utilise those registers, we can use 6 registers for temporary sum and the other 10 can be used to load 40 integers, 20 each from matrixA and matrix. Hence ideally, it should executes 20 times faster than the basic. Using the cycle estimation from section 1.1, the total number of cycles when using NEON could be 98312192/20 ~= 4915610 cycles.

2.2 Compile and Run – NEON SIMD

The results of inline assembly function using NEON are shown in Table 5, and plotted in Figure 6 and Figure 7.

O-level	Perf_Counter_NEON	T_NEON
0	249860900724	208.832216
1	203471277686	170.449745
2	200312510999	167.120374
3	54371739167	45.575891

Table 5: Execution result of inline assembly using NEON SIMD instruction.

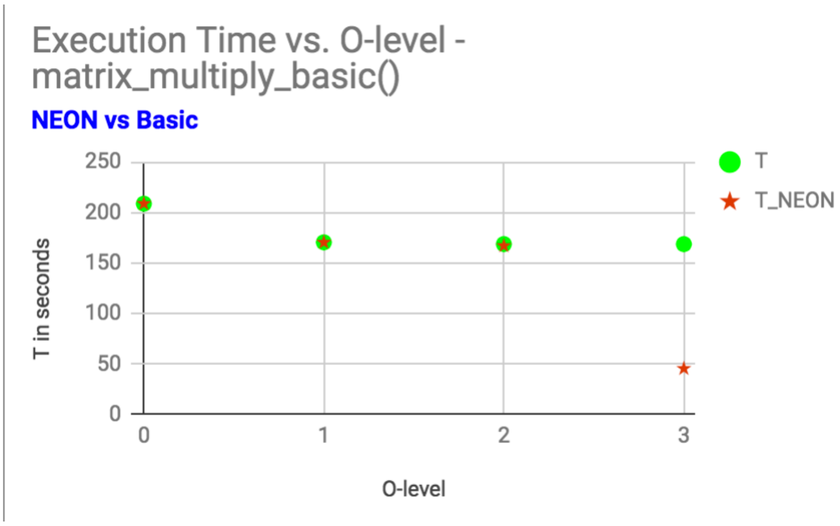

Figure 6: Execution time plot - NEON SIMD vs non-NEON

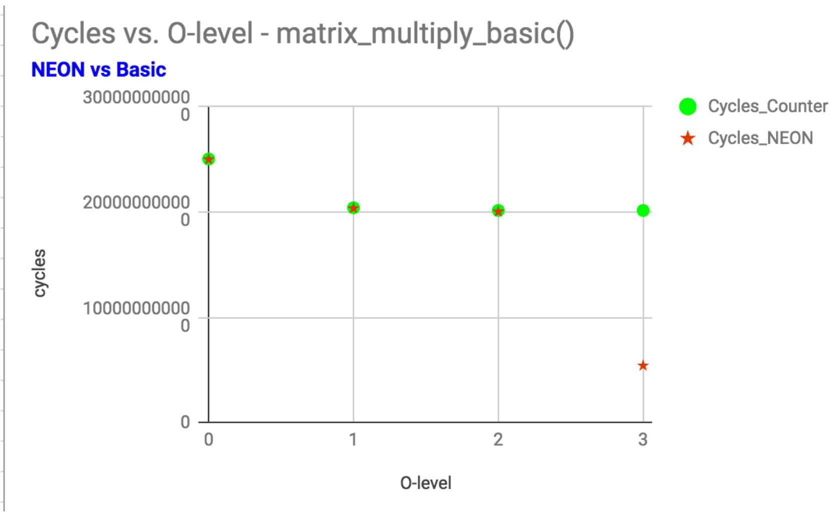

Figure 7: Performance count - NEON SIMD vs. non-NEON

From the experiment results collected, it appears that the compiler only enables the usage of NEON SIMD for O3 optimization. As for O1 and O2, it does not improve the performance.

2.3 Disassemble – Comparison of O0 and O3 – NEON SIMD

This section compares the disassembly codes of matrix_multiply_basic function between O0 and O3 when enabling NEON SIMD, "-mfpu=neon".

```
00010898 <matrix_multiply_basic>:
   10898:	e52db004 	push	{fp}		; (str fp, [sp, #-4]!)
   1089c:	e28db000 	add fp, sp, #0
   108a0:	e24dd014 	sub sp, sp, #20
   108a4:	e3a03000 	mov r3, #0
   108a8:	e50b3008 	str r3, [fp, #-8]
   108ac:	ea00002e 	b   1096c <matrix_multiply_basic+0xd4>
   108b0:	e3a03000 	mov r3, #0
   108b4:	e50b300c 	str r3, [fp, #-12]
   108b8:	ea000025 	b   10954 <matrix_multiply_basic+0xbc>
   108bc:	e3a03000 	mov r3, #0
   108c0:	e50b3010 	str r3, [fp, #-16]
   108c4:	ea00001c 	b   1093c <matrix_multiply_basic+0xa4>
   108c8:	e59f10b8 	ldr r1, [pc, #184]	; 10988 <matrix_multiply_basic+0xf0>
   108cc:	e51b3008 	ldr r3, [fp, #-8]
   108d0:	e1a02503 	lsl r2, r3, #10
   108d4:	e51b300c 	ldr r3, [fp, #-12]
   108d8:	e0823003 	add r3, r2, r3
   108dc:	e7912103 	ldr r2, [r1, r3, lsl #2]
   108e0:	e59f00a4 	ldr r0, [pc, #164]	; 1098c <matrix_multiply_basic+0xf4>
   108e4:	e51b3008 	ldr r3, [fp, #-8]
   108e8:	e1a01503 	lsl r1, r3, #10
   108ec:	e51b3010 	ldr r3, [fp, #-16]
   108f0:	e0813003 	add r3, r1, r3
   108f4:	e7903103 	ldr r3, [r0, r3, lsl #2]
   108f8:	e59fc090 	ldr ip, [pc, #144]	; 10990 <matrix_multiply_basic+0xf8>
   108fc:	e51b1010 	ldr r1, [fp, #-16]
   10900:	e1a00501 	lsl r0, r1, #10
   10904:	e51b100c 	ldr r1, [fp, #-12]
   10908:	e0801001 	add r1, r0, r1
   1090c:	e79c1101 	ldr r1, [ip, r1, lsl #2]
   10910:	e0030391 	mul r3, r1, r3
   10914:	e0822003 	add r2, r2, r3
   10918:	e59f0068 	ldr r0, [pc, #104]	; 10988 <matrix_multiply_basic+0xf0>
   1091c:	e51b3008 	ldr r3, [fp, #-8]
   10920:	e1a01503 	lsl r1, r3, #10
   10924:	e51b300c 	ldr r3, [fp, #-12]
   10928:	e0813003 	add r3, r1, r3
   1092c:	e7802103 	str r2, [r0, r3, lsl #2]
   10930:	e51b3010 	ldr r3, [fp, #-16]
   10934:	e2833001 	add r3, r3, #1
   10938:	e50b3010 	str r3, [fp, #-16]
   1093c:	e51b3010 	ldr r3, [fp, #-16]
   10940:	e3530b01 	cmp r3, #1024	; 0x400
   10944:	baffffdf 	blt 108c8 <matrix_multiply_basic+0x30>
   10948:	e51b300c 	ldr r3, [fp, #-12]
   1094c:	e2833001 	add r3, r3, #1
   10950:	e50b300c 	str r3, [fp, #-12]
   10954:	e51b300c 	ldr r3, [fp, #-12]
   10958:	e3530b01 	cmp r3, #1024	; 0x400
   1095c:	baffffd6 	blt 108bc <matrix_multiply_basic+0x24>
   10960:	e51b3008 	ldr r3, [fp, #-8]
   10964:	e2833001 	add r3, r3, #1
   10968:	e50b3008 	str r3, [fp, #-8]
   1096c:	e51b3008 	ldr r3, [fp, #-8]
   10970:	e3530b01 	cmp r3, #1024	; 0x400
   10974:	baffffcd 	blt 108b0 <matrix_multiply_basic+0x18>
   10978:	e1a00000 	nop			; (mov r0, r0)
   1097c:	e28bd000 	add sp, fp, #0
   10980:	e49db004 	pop {fp}		; (ldr fp, [sp], #4)
   10984:	e12fff1e 	bx  lr
```

Figure 8: Code snippet of the O0 disassembly using "-mfpu=neon" flag

As shown in Figure 8, the O0 optimization did not use any NEON SIMD instruction. Apart from this, it also has four other problems:

- It uses a very limited range of registers, merely {r0-r3}, which in turn results in a relatively frequent use of LDR instruction (as marked in green).
- It has lots of RAW data dependencies. (as underlined in red)
- It compares value with #1024. This does not come free. Better to use 1024 and decrement one by one. Comparing with #0 is more energy efficient than comparing with #1024. (as underlined in blue).
- It does not use more "advanced" instruction, e.g. MLA.

```
000109e4 <matrix_multiply_basic>:
   109e4:   e92d41f0    push    {r4, r5, r6, r7, r8, lr}
   109e8:   e59fe064    ldr lr, [pc, #100]  ; 10a54 <matrix_multiply_basic+0x70>
   109ec:   e59f4064    ldr r4, [pc, #100]  ; 10a58 <matrix_multiply_basic+0x74>
   109f0:   e59f7064    ldr r7, [pc, #100]  ; 10a5c <matrix_multiply_basic+0x78>
   109f4:   e28e8501    add r8, lr, #4194304    ; 0x400000
   109f8:   e59fc060    ldr ip, [pc, #96]   ; 10a60 <matrix_multiply_basic+0x7c>
   109fc:   e1a0600e    mov r6, lr
   10a00:   e1a0500e    mov r5, lr
   10a04:   e1a0300c    mov r3, ip
   10a08:   e28c0501    add r0, ip, #4194304    ; 0x400000
   10a0c:   e1a01004    mov r1, r4
   10a10:   f4652add    vld1.64 {d18-d19}, [r5 :64]!
   10a14:   f4634adf    vld1.64 {d20-d21}, [r3 :64]
   10a18:   e2833a01    add r3, r3, #4096   ; 0x1000
   10a1c:   e1530000    cmp r3, r0
   10a20:   e5b12004    ldr r2, [r1, #4]!
   10a24:   eea02b90    vdup.32 q8, r2
   10a28:   f26029e4    vmla.i32    q9, q8, q10
   10a2c:   1afffff8    bne 10a14 <matrix_multiply_basic+0x30>
   10a30:   e28cc010    add ip, ip, #16
   10a34:   e15c0007    cmp ip, r7
   10a38:   f4462add    vst1.64 {d18-d19}, [r6 :64]!
   10a3c:   1afffff0    bne 10a04 <matrix_multiply_basic+0x20>
   10a40:   e28eea01    add lr, lr, #4096   ; 0x1000
   10a44:   e15e0008    cmp lr, r8
   10a48:   e2844a01    add r4, r4, #4096   ; 0x1000
   10a4c:   1affffe9    bne 109f8 <matrix_multiply_basic+0x14>
   10a50:   e8bd81f0    pop {r4, r5, r6, r7, r8, pc}
   10a54:   00021048    .word   0x00021048
   10a58:   00421044    .word   0x00421044
   10a5c:   00822048    .word   0x00822048
   10a60:   00821048    .word   0x00821048
```

Figure 9: Code snippet of the O3 disassembly using "-mfpu=neon" flag

In Figure 9, the O3 optimization uses NEON SIMD instructions (as underlined in green). Overall, the compiler did a great job in this case. However, it also has the potential to be optimized to get rid of the RAW data dependency issue (as underlined in red) and avoid storing a large number in r8 for comparison (as underlined in blue).

2.4 NEON assembly

```
15 //initialise vectors with random values
16 void init_matrixes(){
17     int i, j, temp;
18     unsigned int p = 0;
19     for (i = 0; i < N; i++) {
20         for (j = 0; j < N; j++){
21             p+= 65610001;            //increment by prime number
22             matrixA[i][j] = p%1336337;  //mod prime number
23             matrixB[i][j] = p%4477457;  //mod prime number
24             matrixC[i][j] = 0;
25         }
26     }
27
28     // transpose B, make it easier for vector vld instruction
29     // if all the request data stays in a row, i.e. in the same cache line
30     for (i = 0; i < N; i++) {
31         for (j = i; j < N; j++) {
32             temp = matrixB[i][j];
33             matrixB[i][j] = matrixB[j][i];
34             matrixB[j][i] = temp;
35         }
36     }
37 }
```

Figure 10:Matrix Transpose

As described in Section 1.1, the bottleneck of the original program is memory access based on the fact that C programming is row majored. In order to make NEON execute faster, the loop is transposed in the init_matrixes function (Line 30 … 36 in Figure 10).

```
72      asm volatile(
73          "push {r0-r12, r14}"              "\n\t"  //save the state of
74                                        // initialise base address regi
75          "mov r3, %[matrixA_addr]"         "\n\t" // matrixA base addr
76          "mov r4, %[matrixB_addr]"         "\n\t" // matrixB base addr
77          "mov r5, %[matrixC_addr]"         "\n\t" // matrixC base addr
78          "mov r12, #4096"                  "\n\t" // frequently used c
79
80          "mov r0, #0"                      "\n\t" // i = 0
81      "I_LOOP:"                             "\n\t"
82          "cmp r0, #1024"                   "\n\t" // Have we reached i
83          "beq END"                         "\n\t" //  - if YES, go to
84                                                   //  - if NO, continu
85          "mla r8, r12, r0, r3"             "\n\t" // moving to NEXT A'
86          "mov r1, #0"                      "\n\t" // j = 0
87      "J_LOOP:"                             "\n\t"
88          "cmp r1, #1024"                   "\n\t" // Have we reached j
89          "beq NEXT_I_ITERATION"            "\n\t" //  - if YES, go to
90                                                   //  - if NO, continu
91          "mov r2, #0"                      "\n\t" // k = 0
92          "mla r10, r12, r1, r4"            "\n\t" // moving to NEXT B'
93          "mov r6, r8"                      "\n\t" // reset the pointer
94          "mov r7, r10"                     "\n\t" // reset the pointer
95      "K_LOOP:"                             "\n\t"
96          "vld1.64 {d0-d3}, [r6:64]!"      "\n\t" // load first 8 eleme
97          "vld1.64 {d4-d7}, [r7:64]!"      "\n\t" // load first 8 eleme
98          "vld1.64 {d8-d11}, [r6:64]!"     "\n\t" // load second 8 elem
99          "vld1.64 {d12-d15}, [r7:64]!"    "\n\t" // load second 8 elem
100                                       // do the vector multiplication
101         "vmul.i32 q8, q0, q4"            "\n\t" // q8 <- q0 * q4
102         "vmul.i32 q9, q1, q5"            "\n\t" // q9 <- q1 * q5
103         "vmul.i32 q10, q2, q6"           "\n\t" // q10 <- q2 * q6
104         "vmul.i32 q11, q3, q7"           "\n\t" // q11 <- q3 * q7
105                                       // reduce everything to d16[0] :
106         "vadd.i32 q10, q10, q11"          "\n\t" // q10 <- q10 + q11
107         "vadd.i32 q8, q8, q9"             "\n\t" // q8 <- q8 + q9
108         "vadd.i32 q8, q8, q10"            "\n\t" // q8 <- q8 + q10
109         "vadd.i32 d16, d16, d17"          "\n\t" // reduce within q8
110         "vpadd.i32 d16, d16"              "\n\t" // reduce within d16
111         "add r2, r2, #1"                  "\n\t" // k++
112         "cmp r2, #64"                     "\n\t" // Have we reached k
113         "bne K_LOOP"                      "\n\t" //   - if NOT, go to
114         "vst1.32 d16[0], [r5:32]!"        "\n\t" //   - if YES, store
115                                                  //     r5, address p
116         "add r1, r1, #1"                  "\n\t" // j++
117         "b J_LOOP"                        "\n\t"
118
119     "NEXT_I_ITERATION:"                   "\n\t"
120         "add r0, r0, #1"                  "\n\t" // i++
121         "b I_LOOP"                        "\n\t" // go to next I loop
122     "END:"                                "\n\t"
123         "pop {r0-r12, r14}"               "\n\t" //retrieve the state
```

Figure 11: NEON inline assembly

As shown in Figure 11, the main idea of implementation here consists of three parts:

- Vector load (Line 96 … 99), processing 16 elements per K iteration.
- Vector multiply (Line 101 … 104) and store the results in temp registers {q8 - q11}.
- Reduce the sum stored in {q8 – q11} (line 106 … 110) to d16[0] in q8.

2.5 NEON Assembly Result

Figure 12 shows the NEON assembly results.

```
pi@raspberrypi:~/lab5_src $ gcc -O3 -mfpu=neon matrix_multiply_neon.c -o matmul_neon
pi@raspberrypi:~/lab5_src $ ./matmul_neon
Running matmul ...
In matmul asm function ...
Execution took 2.139925 seconds.
Performance counter result: 2546922624
pi@raspberrypi:~/lab5_src $
```

Figure 12: NEON assembly results

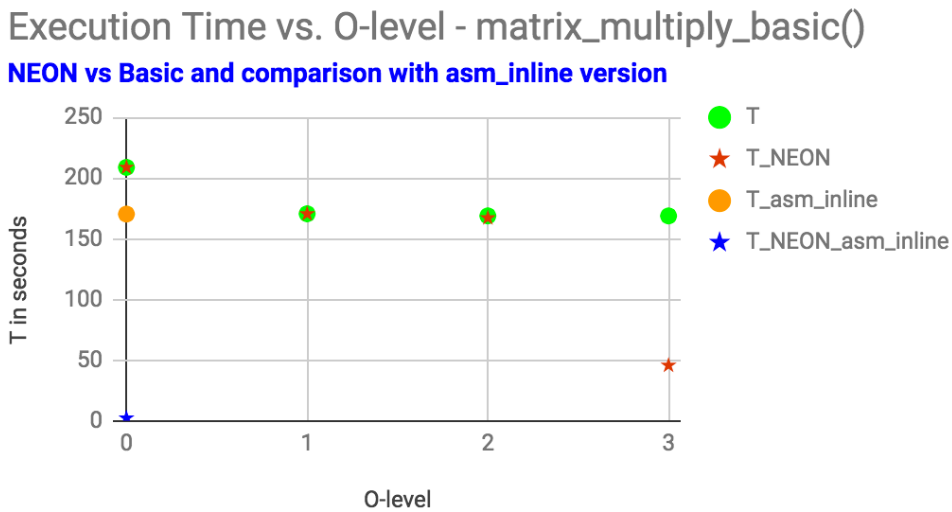

Figure 13: Execution time comparison

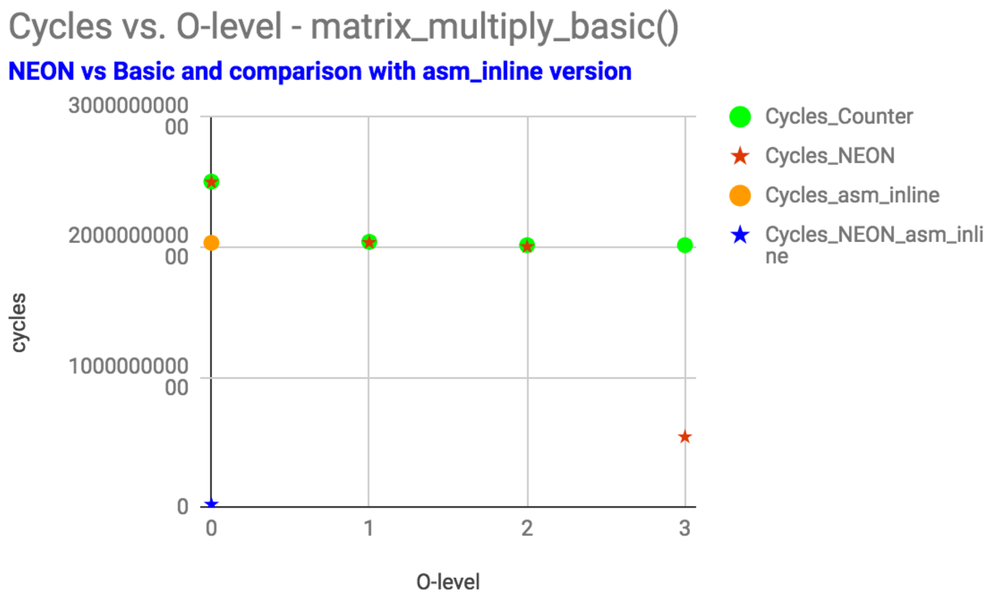

Figure 14: Cycle comparison

Figure 13 and Figure 14 show that the inline assembly (orange circle) can reach the same performance level of O3. NEON inline assembly (blue star) can increase the performance of the program a lot. It can get almost a speedup of 100 and only uses less than 2% cycles compared with the basic function implemented in C.

3 Optimization

The main strategy used to optimize the code includes:

- Matrix transpose of B for better memory access pattern based on the fact that C is row majored.
- NEON SIMD instruction
- Loop unrolling once to allow the processor to issue more independent instructions
- Conditional instructions used, e.g. SUBS and BNE, instead of using CMP r12, #1024
- Prefetching using PLD to notify cache data prefetching of the lines for NEON SIMD VLDR

The results of optimization is shown in Table 6. (For implementation, please refer to the code "*task3_asm_inline_neon_optimized.c*" in the email attachment).

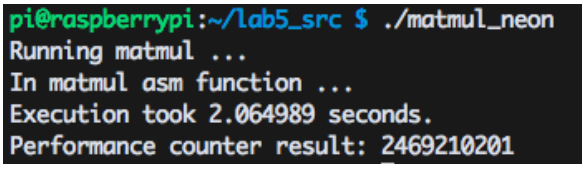

Figure 15: Execution time and performance counter result after optimizaiton.

asm_NEON:		
	Cycles_NEON_asm_inline	T_NEON_asm_inline
	2546922624	2.139925
asm_NEON_Optimize		
	Cycles_NEON_optimize	T_NEON_inline_optimize
	2469210201	2.064989

Table 6: Optimization improves the both cycles and execution time.

To sum up, the final optimized code achieves a significant speedup compared to the original basic function using C:

$$Speedup = \frac{208.806454}{2.064989} = 101.117$$